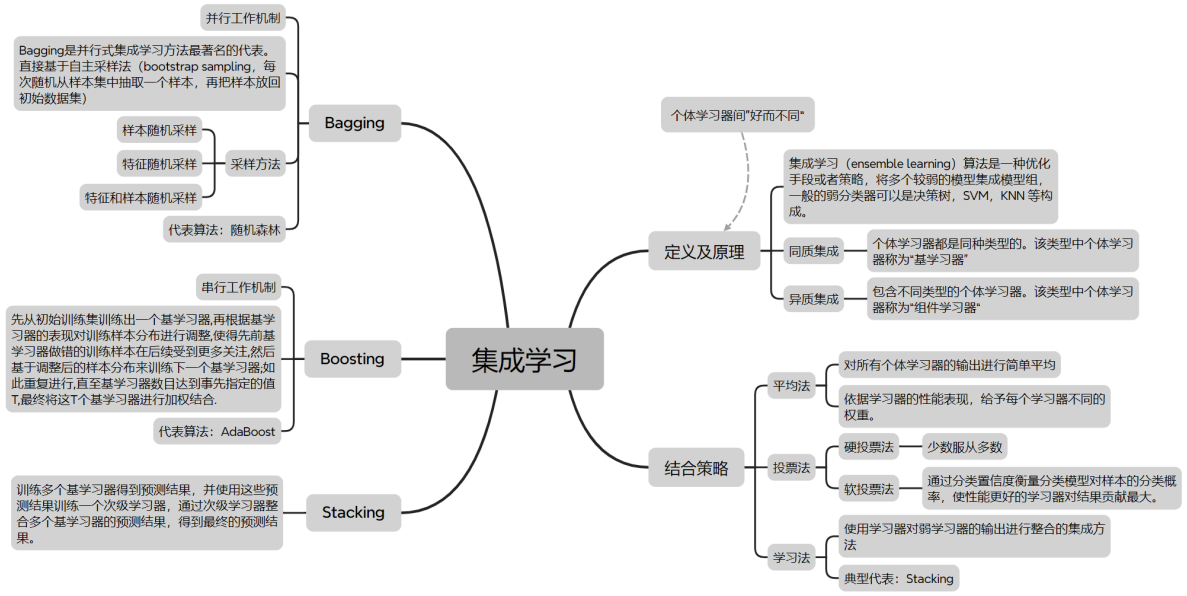
集成学习学习笔记

1. 集成学习基础知识



2. 随机森林 Random Forest

2.1 原理

随机森林(Random Forest, RF)算法是一种基于集成学习的非参数统计学习方法，相比于传统决策树，随机森林通过组合多个决策树的方式实现了对分类或回归问题的更高准确性，但在一定程度上牺牲了模型的解释性。

随机森林的基本思想是通过自助采样和特征随机选择的方式，构建多棵决策树，并通过集成的方式将其组合为一个强分类器，从而提高回归精度。这种策略不仅有效地降低了模型的方差，进一步提高了模型的泛化能力，同时也使模型对异常情况和噪声具有更高的鲁棒性

2.1.1 随机森林的核心思想

随机森林的核心思想是"三个臭皮匠，顶个诸葛亮"。具体可以用两个词概括：**随机**和**森林**。

- **森林**：指的是由多棵决策树组成的集合。每棵树都是一个独立的模型，它们通过投票（分类问题）或平均（回归问题）来决定最终的预测结果。
- **随机**：指的是在构建每棵树时，随机选择样本和特征。这种随机性确保了每棵树都有所不同，避免了过拟合，同时也增加了模型的多样性。

2.1.2 随机森林的流程

随机森林的流程可以简单总结为以下几个步骤：

1. **Bootstrap抽样:** 从原始训练集中随机抽取n个样本（有放回抽样），构成一个新的训练集。该过程重复进行，生成m个不同的训练集。
2. **构建决策树:** 对于每个训练集，使用CART算法构建一棵决策树。在构建过程中，每次分裂节点时，从所有特征中随机选择k个特征，然后选择最优特征进行分裂。
3. **组合预测结果:** 对于新的样本，使用每棵决策树进行预测，并根据任务类型（分类或回归）进行结果整合：

- **分类任务:** 采用投票法，选择得票最多的类别作为最终预测结果。
- **回归任务:** 对所有树的预测结果取平均值作为最终预测结果。

2.1.3 公式推导

1. **分类任务**

   假设随机森林中有T棵决策树，对于一个新的样本x，第i棵树的预测结果为y_i，则随机森林的最终预测结果为：

   $$\hat{y} = \arg\max_{c} \sum_{i=1}^{T} I(y_i = c)$$

   其中，$I(\cdot)$是指示函数，当$y_i = c$时，$I(y_i = c) = 1$，否则为0。

2. **回归任务**

   对于回归任务，随机森林的最终预测结果为所有树的预测结果的平均值：

   $$\hat{y} = \frac{1}{T} \sum_{i=1}^{T} y_i$$

2.2 优缺点

- **优点:**
  - 泛化能力强，能够有效处理高维数据。
  - 对缺失值和异常值不敏感。
  - 能够评估特征的重要性。
- **缺点:**
  - 与决策树相比，模型解释性较差。
  - 训练时间较长，尤其是当数据量较大时。

2.3 代码实战

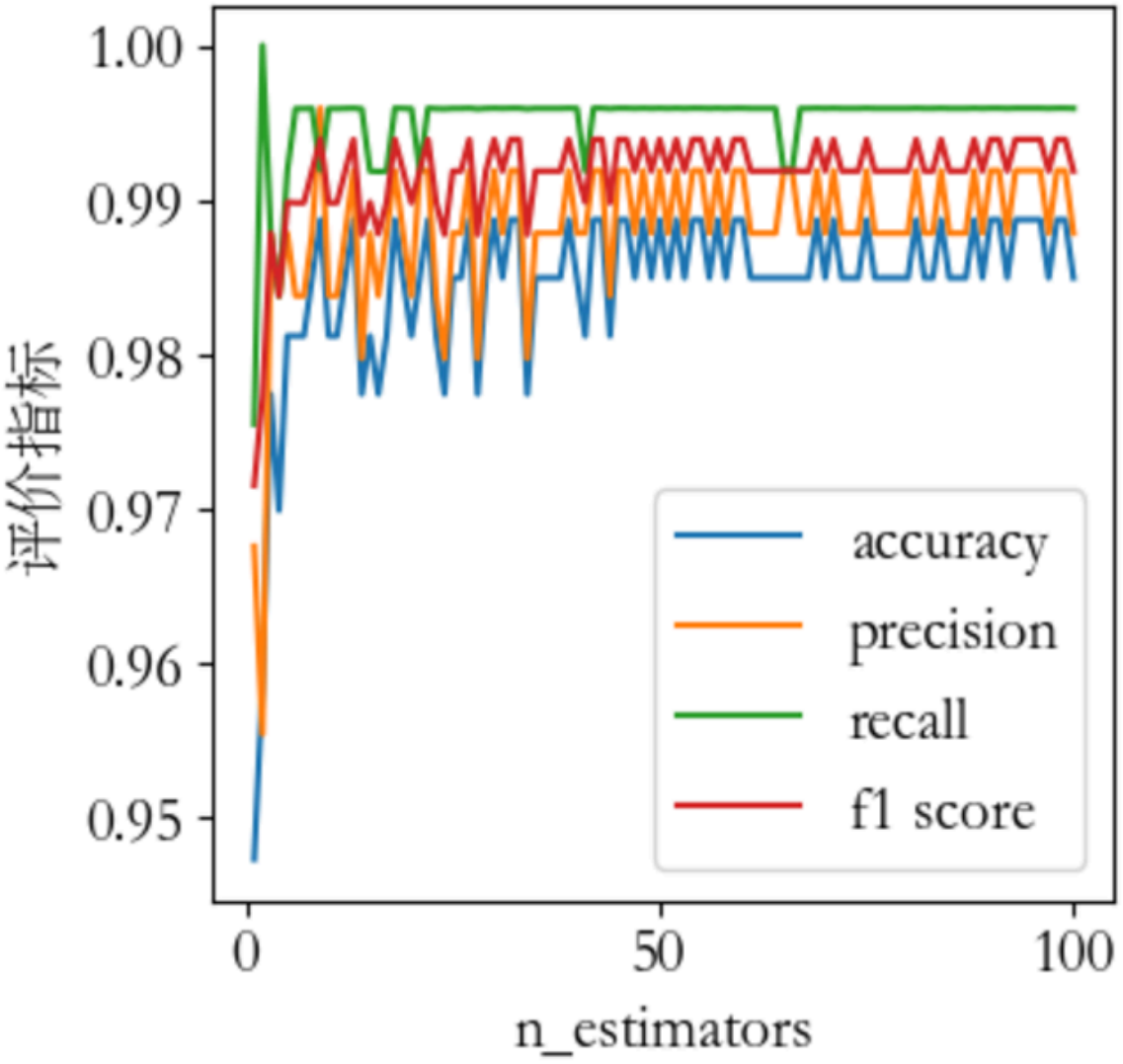
2.3.1 关键参数

- **n_estimators:** 随机森林中决策树的数量。一般来说，树的数量越多，模型的性能越好，但计算成本也越高。
- **max_features:** 每次分裂节点时，随机选择的特征数量。该参数控制着树的多样性和相关性，通常设置为特征总数的平方根。
- **max_depth:** 决策树的最大深度。限制树的深度可以防止过拟合，但可能会导致欠拟合。

2.3.2 基于随机森林算法的银行危机预测

基于材料中给定的案例，运行随机森林预测代码，结果如下：

```
***** 决策树 *****
accuracy 0.9584905660377359
precision 0.9634146341463414
recall 0.9916317991631799
f1 score 0.977319587628866
***** 随机森林 *****
accuracy 0.9849056603773585
precision 0.9878048780487805
recall 0.9959016393442623
f1 score 0.9918367346938776
```



与决策树算法相比，随机森林算法在准确率、精确率、召回率和F1上均具有更好的效果，说明通过bagging集成多棵决策树的预测性能优于单棵决策树。随着决策树数量的增加，模型的性能指标逐渐提高并趋于稳定。这表明增加决策树的数量可以提高模型的性能，但当达到一定数量后，性能提升将不再显著。

然而与决策树相比，随机森林算法的可解释性更低，无法直观判断对结果影响更大的因素。因此，对案例进行扩展，输出随机森林算法特征重要性排名。

2.4 扩展

2.4.1 特征重要性

特征重要性是嵌入在随机森林算法内部的解释方式，通常使用采样过程中袋外数据（out of bag, OOB）进行度量表示，以评估每个特征对模型预测能力的贡献。具体上，随机交换袋外中某个特征的值，然后进行重新预测，回归的准确度下降程度即为该特征的重要性。在包含T棵决策树的随机森林模型上，数据集(X, y) 的第j个特征的重要性表示为：

$$MDA(j) = \frac{1}{T}\sum_{t}^{T}\left[\frac{1}{|D_t|}\left(\sum_{X_i^j \in D_t^j}\sum_{k}\left(R_k(X_i^j) - y_i^k\right)^2 - \sum_{X_i \in D_t}\sum_{k}\left(R_k(X_i) - y_i^k\right)^2\right)\right].$$

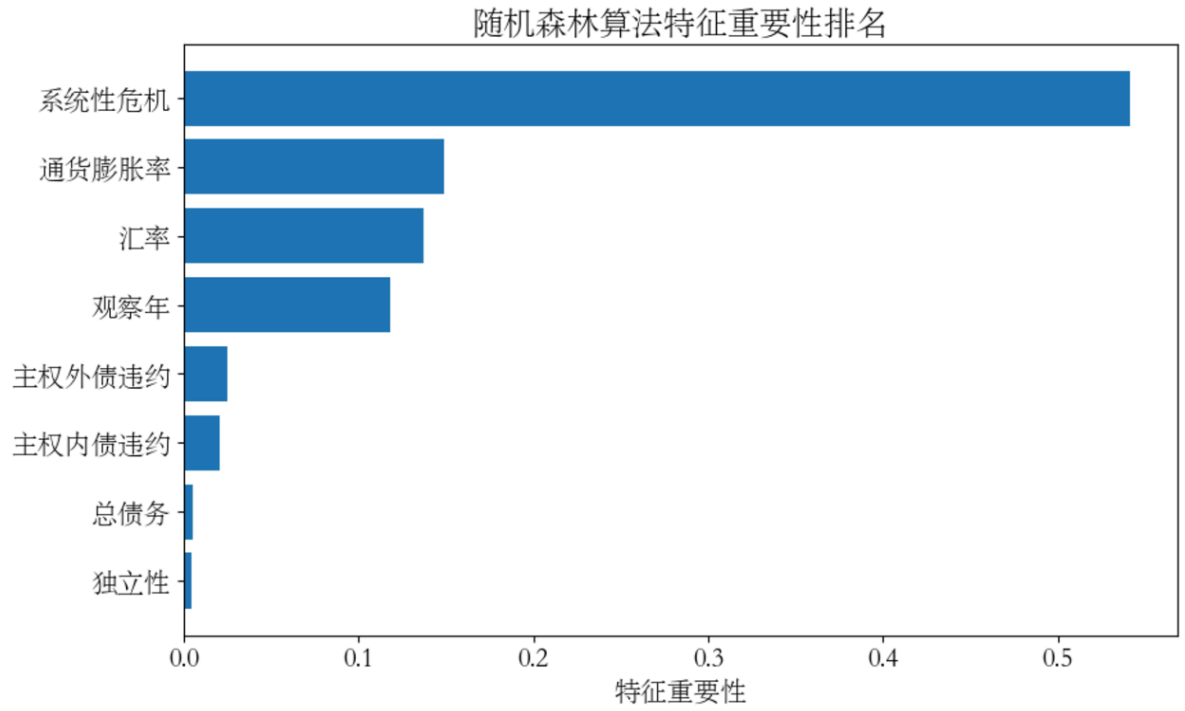
其中，X_i^j表示X_i的第j维特征被交换后的样本，D_t表示决策树t的袋外数据集，D_t^j表示第j维交换后形成的样本集，$R(X_i)$为样本X_i的预测输出，y_i^k考虑到多目标回归，表示第k维的输出。

2.4.2 特征重要性案例

使用".feature_importances_"获得特征重要性，输出特征重要性及其排名。完整代码如下：

```
# 输出随机森林算法的特征重要性
importances = rf_clf.feature_importances_
indices = np.argsort(importances)[::-1]

# 获取特征名称
features = data.drop('银行危机', axis=1).columns

# 打印特征及其重要性
print("特征重要性排名：")
for f in range(len(indices)):
    print(f"{f + 1}. {features[indices[f]]}: {importances[indices[f]]:.4f}")

# 绘制特征重要性直方图
plt.figure(figsize=(10, 6))
plt.barh(range(len(indices)), importances[indices], align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('特征重要性')
plt.title('随机森林算法特征重要性排名')
plt.gca().invert_yaxis()  # 反转y轴，使最重要的特征在顶部
plt.show()
```

```
特征重要性排名:
1. 系统性危机: 0.5412
2. 通货膨胀率: 0.1486
3. 汇率: 0.1372
4. 观察年: 0.1179
5. 主权外债违约: 0.0251
6. 主权内债违约: 0.0208
7. 总债务: 0.0052
8. 独立性: 0.0041
```



从特征重要性结果可以发现“系统性危机”对于结果的贡献程度为0.5412，“总债务”和“独立性”对于结果的贡献度很小。可以基于特征重要性结果，对特征进行筛选。

删掉“总债务”和“独立性”两个指标后，再次对银行危机进行预测，结果如下：

```
***** 决策树 *****
accuracy 0.9660377358490566
precision 0.967479674796748
recall 0.99581589958159
f1 score 0.9814432989690721
***** 随机森林 *****
accuracy 0.9849056603773585
precision 0.9878048780487805
recall 0.9959016393442623
f1 score 0.9918367346938776
```

从结果可以发现，删除这两个指标后，决策树的各评价指标均有提升，随机森林的评价指标基本保持不变。由此可以得出，“总债务”和“独立性”这两个指标为冗余指标，可以删除。同时，随机森林可以作为一种特征筛选方法对特征进行筛选，提升决策结果和效率。

2.5 总结

随机森林是一个强大且易于使用的算法，尤其适合处理高维数据和大规模数据集。它的随机性和集成思想使得它在很多场景下都能表现出色。虽然它的解释性较差，但在实际应用中，我们往往更关注预测的准确性。同时，随机森林的特征重要性使得随机森林在某些场景下可以作为特征筛选方法进行使用，帮助我们识别和选择对模型预测最有影响的特征。虽然近年来出现了一些性能更优的算法，如XGBoost、LightGBM等，但随机森林因其原理简单、易于实现、性能出色、适用场景广泛等特点，仍然在工业界和学术界得到了广泛的应用。

3. AdaBoost

3.1原理

AdaBoost（Adaptive Boosting）是一种经典的集成学习算法，通过组合多个弱分类器来构建一个强分类器。与随机森林的Bagging思想不同，AdaBoost采用的是Boosting思想，即通过逐步调整样本权重，使得模型能够专注于那些难以分类的样本，从而提高整体模型的性能。

3.1.1 AdaBoost的核心思想

AdaBoost的核心思想是**“分而治之”**，即通过逐步调整样本的权重，使得每一轮训练都更加关注上一轮分类错误的样本，最终将所有弱分类器的结果加权组合成一个强分类器。具体来说，AdaBoost的每一轮训练都会根据上一轮的结果调整样本的权重，使得分类错误的样本在下一轮中获得更高的权重，从而让模型更加关注这些难以分类的样本。

3.1.2 AdaBoost的流程

AdaBoost的流程可以总结为以下几个步骤：

1. **初始化样本权重**：假设训练集有N个样本，初始时每个样本的权重为$w_i = \frac{1}{N}$。
2. **训练弱分类器**：在每一轮训练中，使用当前的样本权重训练一个弱分类器$G_m(x)$。
3. **计算分类误差率**：计算当前弱分类器的分类误差率 e_m：

$$e_m = \sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))$$

   其中，$I(\cdot)$是指示函数，当$y_i \neq G_m(x_i)$时，$I(y_i \neq G_m(x_i)) = 1$，否则为0。

4. **计算弱分类器的权重**：根据分类误差率e_m，计算当前弱分类器的权重α_m：

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - e_m}{e_m} \right)$$

5. **更新样本权重**：根据当前弱分类器的表现，更新样本的权重：

$$w_i \leftarrow w_i \cdot \exp\left(-\alpha_m y_i G_m(x_i)\right)$$

   然后对权重进行归一化，使得$\sum_{i=1}^{N} w_i = 1$。

6. **组合弱分类器**：将所有弱分类器的结果加权组合，得到最终的强分类器：

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x) \right)$$

3.1.3 公式推导

1. **弱分类器的权重α_m的推导**

   弱分类器的权重α_m是通过最小化指数损失函数得到的。假设我们有一个弱分类器$G_m(x)$，其分类误差率为 e_m，则指数损失函数可以表示为：

$$L(\alpha_m) = \sum_{i=1}^{N} w_i \exp\left(-\alpha_m y_i G_m(x_i)\right)$$

   为了最小化损失函数，我们对 α_m求导并令导数为0：

$$\frac{\partial L(\alpha_m)}{\partial \alpha_m} = -\sum_{i=1}^{N} w_i y_i G_m(x_i) \exp\left(-\alpha_m y_i G_m(x_i)\right) = 0$$

通过求解可以得到：

$$\alpha_m = \frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right)$$

3.1.4 AdaBoost的特点

- **自适应**：AdaBoost通过逐步调整样本权重，使得模型能够自适应地关注那些难以分类的样本。
- **弱分类器的组合**：AdaBoost通过加权组合多个弱分类器，能够显著提升模型的性能。
- **对异常值敏感**：由于AdaBoost会不断调整样本权重，异常值可能会在训练过程中获得较高的权重，从而影响模型的性能。

3.2 优缺点

- **优点:**
  - 能够显著提升弱分类器的性能。
  - 对高维数据和小样本数据表现良好。
  - 不需要对弱分类器进行复杂的调参。
- **缺点:**
  - 对噪声和异常值敏感。
  - 训练时间较长，尤其是在弱分类器较复杂时。

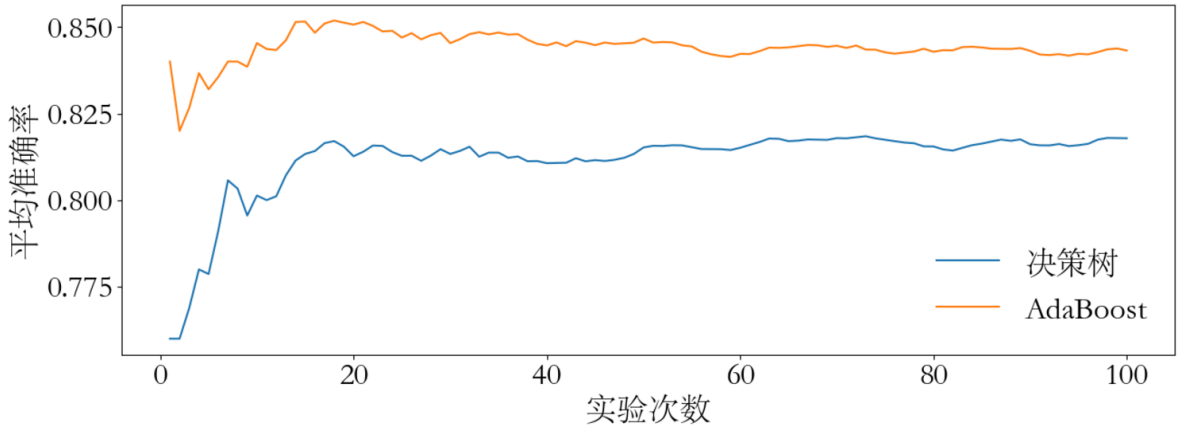
3.3 代码实战

3.3.1 关键参数

- **n_estimators:** AdaBoost中弱分类器的数量。一般来说，弱分类器的数量越多，模型的性能越好，但计算成本也越高。
- **learning_rate:** 学习率，控制每个弱分类器对最终模型的贡献程度。较小的学习率需要更多的弱分类器来达到相同的性能。
- **estimator:** 弱分类器的类型，默认是决策树。

3.3.2 基于AdaBoost算法的马疝病预测

基于材料中给定的案例，运行AdaBoost预测代码，结果如下：




```
决策树平均准确率： 0.8179
AdaBoost平均准确率： 0.8432
```

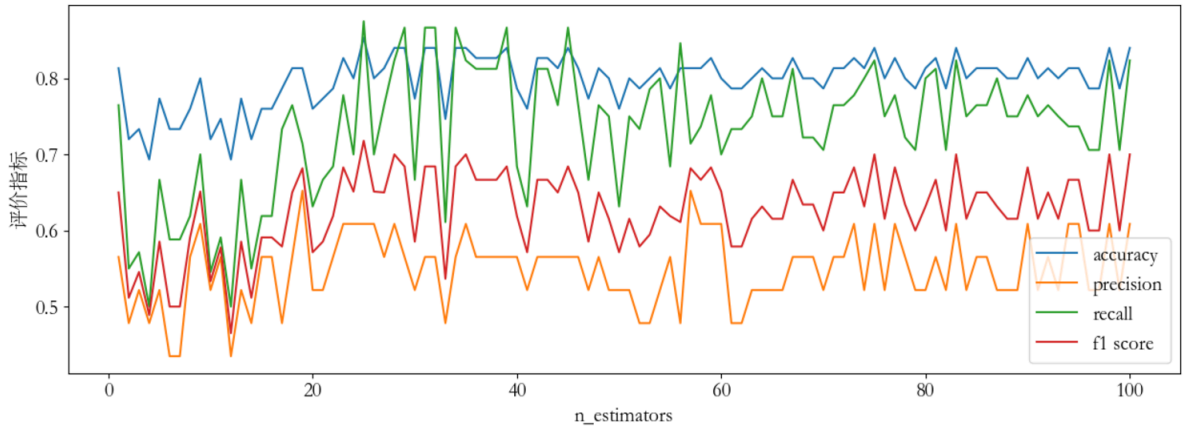
在不同的数据子集下进行重复实验，可以看到随着实验次数的增加，模型性能逐渐趋于稳定，并且AdaBoost算法的模型效果由于决策树。

3.4 扩展

3.4.1 n_estimators参数变化影响

变化参数n_estimators，观察其对AdaBoost模型影响，代码如下：

```
# 绘制n_estimators参数对预测性能的影响曲线
accuracy_list, precision_list, recall_list, f1_list = [], [], [], []
for n in range(1,101):
    clf =
ensemble.AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=7,
min_samples_leaf=7), n_estimators=n, algorithm='SAMME', learning_rate=0.95)  # 构
建随机森林模型
    clf.fit(X_train, y_train)
    y_pred_rf = clf.predict(X_test)
    accuracy_list.append(accuracy_score(y_pred_rf, y_test))
    precision_list.append(precision_score(y_pred_rf, y_test))
    recall_list.append(recall_score(y_pred_rf, y_test))
    f1_list.append(f1_score(y_pred_rf, y_test))

# 5.绘制性能曲线
plt.rcParams['font.sans-serif']=['STSong']
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(15,5))
plt.plot(list(range(1, 101)), accuracy_list, label='accuracy')
plt.plot(list(range(1, 101)), precision_list, label='precision')
plt.plot(list(range(1, 101)), recall_list, label='recall')
plt.plot(list(range(1, 101)), f1_list, label='f1 score')
plt.xlabel('n_estimators')
plt.ylabel('评价指标')
plt.legend()
plt.show()
```

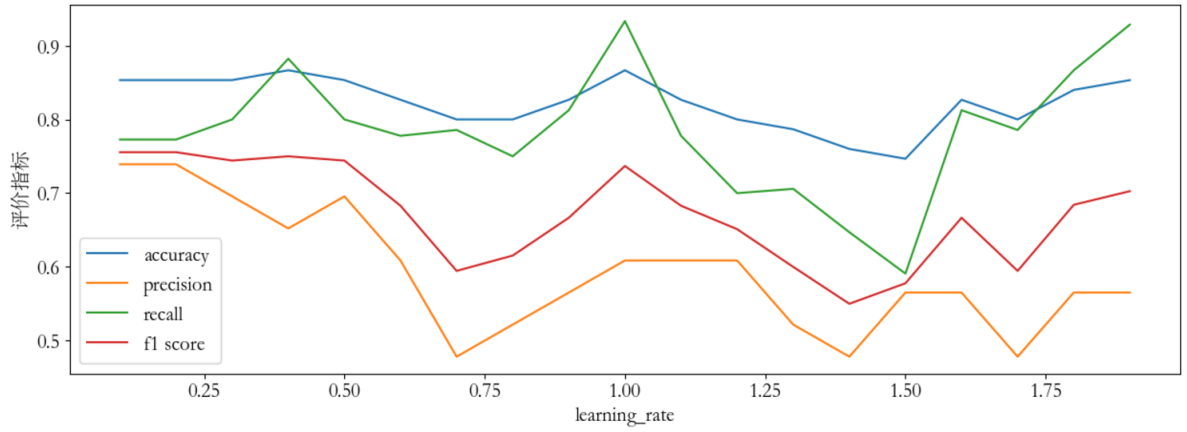


从结果中可以看出，与随机森林随n_estimators变化模型性能趋于稳定不同，AdaBoost模型性能随n_estimators变化呈现不规律变动，且不同n_estimators参数对模型影响较大。因此在实际使用AdaBoost进行预测时，需要根据实际问题进行模型调参。

3.4.2 learning_rate参数变化影响

变化参数learning_rate，观察其对AdaBoost模型影响，代码如下：

```
# 绘制learning_rate参数对预测性能的影响曲线
accuracy_list, precision_list, recall_list, f1_list = [], [], [], []
for l in range(1, 20):
    clf =
ensemble.AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=7,
min_samples_leaf=7), n_estimators=100, algorithm='SAMME', learning_rate=l*0.1)  #
构建随机森林模型
    clf.fit(X_train, y_train)
    y_pred_rf = clf.predict(X_test)
    accuracy_list.append(accuracy_score(y_pred_rf, y_test))
    precision_list.append(precision_score(y_pred_rf, y_test))
    recall_list.append(recall_score(y_pred_rf, y_test))
    f1_list.append(f1_score(y_pred_rf, y_test))

# 5.绘制性能曲线
plt.rcParams['font.sans-serif']=['STSong']
plt.rcParams.update({'font.size': 15})
plt.figure(figsize=(15,5))
plt.plot(list(np.arange(0.1, 2, 0.1)), accuracy_list, label='accuracy')
plt.plot(list(np.arange(0.1, 2, 0.1)), precision_list, label='precision')
plt.plot(list(np.arange(0.1, 2, 0.1)), recall_list, label='recall')
plt.plot(list(np.arange(0.1, 2, 0.1)), f1_list, label='f1 score')
plt.xlabel('learning_rate')
plt.ylabel('评价指标')
plt.legend()
plt.show()
```



从结果中可以看出，AdaBoost模型性能随learning_rate变化呈现不规律变动，且不同learning_rate参数对模型影响较大。learning_rate为1时，模型效果最好。因此在实际使用AdaBoost进行预测时，需要根据实际问题进行模型调参。

3.5 总结

AdaBoost是一种强大的集成学习算法，通过逐步调整样本权重，能够显著提升弱分类器的性能。虽然它对噪声和异常值较为敏感，但在处理高维数据和小样本数据时表现优异。通过合理调参，AdaBoost可以在许多分类任务中取得出色的结果。

4. GBDT

4.1 原理

GBDT（Gradient Boosting Decision Tree，梯度提升决策树）是一种基于Boosting思想的集成学习算法。与AdaBoost不同，GBDT通过梯度下降的方式逐步优化模型，每一轮训练都试图减少前一轮模型的残差（即预测误差），最终将所有弱学习器的结果累加，形成一个强学习器。

GBDT的核 心思想是**“分步优化”**，即通过逐步拟合残差，使得模型能够逐步逼近真实值。GBDT的每一轮训练都会构建一棵新的决策树，用于拟合前一轮模型的残差，最终将所有树的结果累加，得到最终的预测结果。

4.1.1 GBDT的核心思想

GBDT的核心思想可以概括为以下几点：

- **残差拟合**：每一轮训练的目标是拟合前一轮模型的残差（即真实值与预测值之间的差异）。
- **梯度下降**：GBDT通过梯度下降的方式优化模型，每一轮训练都试图减少损失函数的梯度。
- **累加模型**：最终模型是所有弱学习器的累加结果。

4.1.2 GBDT的流程

GBDT的流程可以总结为以下几个步骤：

1. **初始化模型**：假设初始模型为一个常数，通常为训练集目标值的均值：

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$

   其中，$L(y_i, \gamma)$是损失函数，γ是常数。

2. **迭代训练**：对于每一轮$m = 1, 2, \ldots, M$，执行以下步骤：
   - **计算残差**：计算当前模型的残差：

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

     其中，r_{im}是第i个样本在第 m轮的残差。
   - **拟合残差**：使用决策树拟合残差 r_{im}，得到第m棵树$h_m(x)$。
   - **更新模型**：将新树的结果累加到当前模型中：

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x)$$

     其中，ν是学习率，用于控制每棵树的贡献程度。

3. **输出最终模型**：经过 M*M* 轮训练后，最终的模型为：

$$F(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

4.1.3 公式推导

1. **梯度下降法**

GBDT通过梯度下降的方式优化模型。假设损失函数为$L(y_i, F(x_i))$，则第轮m的残差可以通过损失函数的负梯度计算得到：

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

例如，对于平方损失函数$L(y_i, F(x_i)) = \frac{1}{2}(y_i - F(x_i))^2$，其梯度为：

$$r_{im} = y_i - F_{m-1}(x_i)$$

4.2 优缺点

- **优点:**
  - 能够处理非线性关系，适合复杂的数据集。
  - 可以自定义损失函数，灵活性高。
  - 在许多任务中表现优异，尤其是在结构化数据上。
  - 可以用于回归、分类和排序任务，具有很高的灵活性。
- **缺点:**
  - 训练时间较长，尤其是在数据量较大时。
  - 对异常值敏感，容易过拟合。
  - 需要仔细调参，尤其是学习率和树的数量。

4.3 代码实战

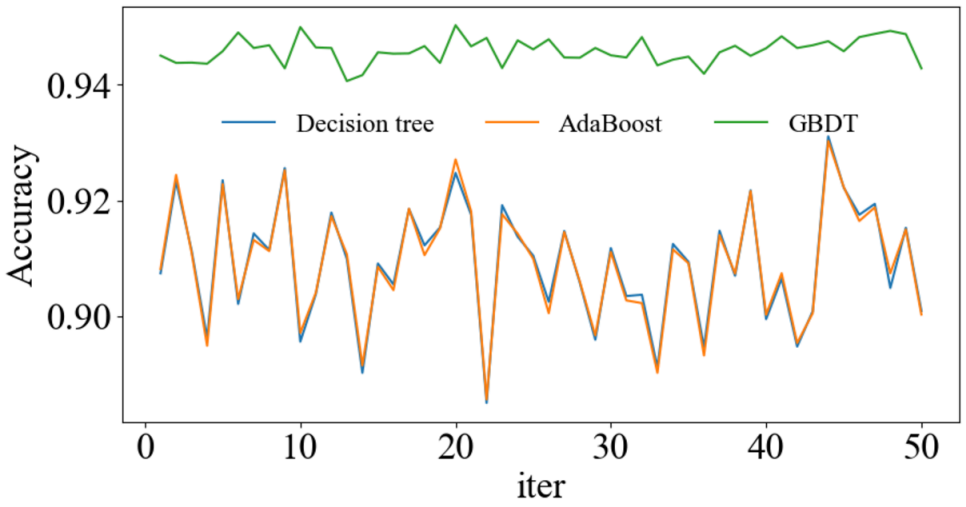
4.1.3 关键参数

- **n_estimators:** GBDT中树的数量。树的数量越多，模型的性能越好，但计算成本也越高。
- **learning_rate:** 学习率，控制每棵树的贡献程度。较小的学习率需要更多的树来达到相同的性能。
- **max_depth:** 每棵树的最大深度。限制树的深度可以防止过拟合。
- **loss:** 损失函数，默认是平方损失函数（用于回归）或对数损失函数（用于分类）。

4.3.2 基于梯度提升数算法的充电桩故障状态预测

基于材料中给定的案例，运行AdaBoost预测代码，结果如下：

```
label
1    42750
0    42750
Name: count, dtype: int64
100%|██████████████████████████████████████████████████████████████████████████| 50/50 [12:01<00:00, 14.43s/it]
```




```
决策树的平均准确率： 0.9093660818713449
AdaBoost的平均准确率： 0.9092397660818715
梯度提升树的平均准确率： 0.9458498245614034
```

通过多次重复实验来验证模型的稳定性，与决策树和AdaBoost进行对比，从结果可以发现，GBDT表现出比决策树和AdaBoost更高的准确率和稳定性。

4.4 扩展

4.4.1 基于GBDT的回归任务

利用GBDT算法对模拟数据进行回归。

模型构建。在Python中，GBDT模型通过GradientBoostingRegressor类实现，代码如下：gbdt_reg = GradientBoostingRegressor()。

完整代码如下：

```
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor,
AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 生成回归数据集
X, y = make_regression(n_samples=1000, n_features=10, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 初始化模型
gbdt_reg = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
max_depth=3, random_state=42)
rf_reg = RandomForestRegressor(n_estimators=100, max_depth=3, random_state=42)
dt_reg = DecisionTreeRegressor(max_depth=3, random_state=42)
ada_reg = AdaBoostRegressor(n_estimators=100, learning_rate=0.1, random_state=42)

# 训练模型
gbdt_reg.fit(X_train, y_train)
rf_reg.fit(X_train, y_train)
dt_reg.fit(X_train, y_train)
```

```
ada_reg.fit(X_train, y_train)

# 预测
y_pred_gbdt = gbdt_reg.predict(X_test)
y_pred_rf = rf_reg.predict(X_test)
y_pred_dt = dt_reg.predict(X_test)
y_pred_ada = ada_reg.predict(X_test)

# 评估模型
mse_gbdt = mean_squared_error(y_test, y_pred_gbdt)
mse_rf = mean_squared_error(y_test, y_pred_rf)
mse_dt = mean_squared_error(y_test, y_pred_dt)
mse_ada = mean_squared_error(y_test, y_pred_ada)

print(f"GBDT回归器的均方误差: {mse_gbdt:.4f}")
print(f"随机森林回归器的均方误差: {mse_rf:.4f}")
print(f"决策树回归器的均方误差: {mse_dt:.4f}")
print(f"AdaBoost回归器的均方误差: {mse_ada:.4f}")
```

```
GBDT回归器的均方误差: 1321.5928
随机森林回归器的均方误差: 7004.3019
决策树回归器的均方误差: 9341.6678
AdaBoost回归器的均方误差: 5852.4676
```

从结果中可以发现，对于随机数据，GBDT回归器的均方误差为1321.59，与随机森林、决策树和AdaBoost回归器相比效果更好。

4.4.2 算例计算

通过一个简单的示例说明GBDT的计算过程：

假设我们有一个简单的数据集，包含两个特征 x_1 和x_2，以及目标变量 y：

***x*1**	***x*2**	***y***
1	1	2
2	0	3
3	1	5
4	0	6
5	1	8

我们的目标是使用GBDT来预测y的值。

初始化

1. **初始化模型**：首先，我们初始化模型$F_0(x)$ 为常数模型，通常取y的平均值。在这个例子中，y的平均值是 $(2+3+5+6+8)/5=4.8$。
2. **计算残差**：计算每个数据点的残差r_{im}，即实际值与模型预测值之间的差异。对于 $F_0(x)=4.8$，残差为：
    - $r_{i1}=2-4.8=-2.8$

- $r_{i2} = 3 - 4.8 = -1.8$
- $r_{i3} = 5 - 4.8 = 0.2$
- $r_{i4} = 6 - 4.8 = 1.2$
- $r_{i5} = 8 - 4.8 = 3.2$

第一棵树

1. **拟合残差**：我们构建第一棵树 $h_1(x)$ 来拟合这些残差。假设 $h_1(x)$是一个简单的决策树，它根据 x_1的值进行分裂：
   - 如果 $x_1 < 2.5$，则 $h_1(x) = -1$
   - 否则，$h_1(x) = 1$
2. **更新模型**：更新模型 $F_1(x) = F_0(x) + \nu \cdot h_1(x)$，其中 ν是学习率，通常取一个小值如 0.1。因此，$F_1(x) = 4.8 + 0.1 \cdot h_1(x)$。
3. **计算新的残差**：计算新的残差 r_{i2}：
   - 对于 $x_1 = 1$，$h_1(x) = -1$，新的残差 $r_{i1} = -2.8 - 0.1 \cdot (-1) = -2.7$
   - 对于 $x_1 = 2$，$h_1(x) = 1$，新的残差 $r_{i2} = -1.8 - 0.1 \cdot 1 = -1.9$
   - 对于 $x_1 = 3$，$h_1(x) = 1$，新的残差 $r_{i3} = 0.2 - 0.1 \cdot 1 = 0.1$
   - 对于 $x_1 = 4$，$h_1(x) = 1$，新的残差 $r_{i4} = 1.2 - 0.1 \cdot 1 = 1.1$
   - 对于 $x_1 = 5$，$h_1(x) = 1$，新的残差 $r_{i5} = 3.2 - 0.1 \cdot 1 = 3.1$

第二棵树

1. **拟合新的残差**：构建第二棵树 $h_2(x)$来拟合新的残差。假设 $h_2(x)$根据 x_2的值进行分裂：
   - 如果 $x_2 < 0.5$，则 $h_2(x) = -0.5$
   - 否则，$h_2(x) = 0.5$
2. **更新模型**：更新模型 $F_2(x) = F_1(x) + \nu \cdot h_2(x)$。因此，$F_2(x) = 4.8 + 0.1 \cdot h_1(x) + 0.1 \cdot h_2(x)$。
3. **计算新的残差**：计算新的残差 r_{i3}：
   - 对于 $x_1 = 1, x_2 = 1$，$h_1(x) = -1$，$h_2(x) = 0.5$，新的残差 $r_{i1} = -2.7 - 0.1 \cdot 0.5 = -2.75$
   - 对于 $x_1 = 2, x_2 = 0$，$h_1(x) = 1$，$h_2(x) = -0.5$，新的残差 $r_{i2} = -1.9 - 0.1 \cdot (-0.5) = -1.85$
   - 对于 $x_1 = 3, x_2 = 1$，$h_1(x) = 1$，$h_2(x) = 0.5$，新的残差 $r_{i3} = 0.1 - 0.1 \cdot 0.5 = 0.05$
   - 对于 $x_1 = 4, x_2 = 0$，$h_1(x) = 1$，$h_2(x) = -0.5$，新的残差 $r_{i4} = 1.1 - 0.1 \cdot (-0.5) = 1.15$
   - 对于 $x_1 = 5, x_2 = 1$，$h_1(x) = 1$，$h_2(x) = 0.5$，新的残差 $r_{i5} = 3.1 - 0.1 \cdot 0.5 = 3.05$

继续迭代

重复上述过程，直到达到预定的树的数量或残差足够小。每一步都试图通过拟合残差来改进模型。

最终模型

最终模型 $F(x)$是所有树的加权和：

$$F(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

这个例子展示了GBDT的基本过程，每一步都通过添加新的树来逐步改进模型。在实际应用中，GBDT可以处理更复杂的数据集和更多的特征。

4.5 总结

GBDT是一种强大的集成学习算法，通过逐步拟合残差和梯度下降的方式优化模型，能够处理复杂的非线性关系。虽然它对异常值敏感且训练时间较长，但在许多任务中表现优异，尤其是在结构化数据上。通过合理调参，GBDT可以在回归、分类和排序任务中取得出色的结果。

5. Xgboost

5.1 原理

XGBoost（eXtreme Gradient Boosting）是GBDT（Gradient Boosting Decision Tree）的一个高效实现，由陈天奇于2014年提出。XGBoost在GBDT的基础上进行了大量优化，包括正则化、并行计算、缺失值处理等，使其在大规模数据集上表现出色，成为许多机器学习竞赛中的“神器”。

5.1.1 XGBoost的核心思想

XGBoost的核心思想仍然是**“分步优化”**，即通过逐步拟合残差来优化模型。但与GBDT不同的是，XGBoost在目标函数中引入了正则化项，并通过二阶泰勒展开近似损失函数，从而加速模型的训练过程。

- **正则化**：在目标函数中加入正则化项，控制模型的复杂度，防止过拟合。
- **二阶泰勒展开**：通过二阶泰勒展开近似损失函数，加速模型的优化过程。
- **并行计算**：通过特征并行和数据并行，提升模型的训练速度。
- **缺失值处理**：自动处理缺失值，无需额外预处理。

5.1.2 XGBoost的流程

XGBoost的流程可以总结为以下几个步骤：

1. **初始化模型**：假设初始模型为一个常数，通常为训练集目标值的均值：

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$

其中，$L(y_i, \gamma)$是损失函数，γ是常数。

2. **迭代训练**：对于每一轮$m = 1, 2, \ldots, M$，执行以下步骤：
   - **计算一阶和二阶梯度**：计算当前模型的一阶梯度 g_i和二阶梯度 h_i：

$$g_i = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}, \quad h_i = \frac{\partial^2 L(y_i, F(x_i))}{\partial F(x_i)^2}$$

   - **构建决策树**：使用贪心算法构建决策树，选择最优的分裂点，使得目标函数最小化：

$$\text{Gain} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

其中，I_L 和 I_R 分别是分裂后的左子树和右子树的样本集合，λ 和 γ 是正则化参数。

- **更新模型**：将新树的结果累加到当前模型中：

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x)$$

其中，ν 是学习率，用于控制每棵树的贡献程度。

3. **输出最终模型**：经过 M 轮训练后，最终的模型为：

$$F(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

5.1.3 公式推导

1. **目标函数**

XGBoost的目标函数由损失函数和正则化项组成：

$$\text{Obj} = \sum_{i=1}^{N} L(y_i, F(x_i)) + \sum_{m=1}^{M} \Omega(h_m)$$

其中，$\Omega(h_m)$ 是正则化项，用于控制模型的复杂度：

$$\Omega(h_m) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

其中，T 是树的叶子节点数，w_j 是第 j 个叶子节点的权重。

2. **二阶泰勒展开**

为了加速优化过程，XGBoost使用二阶泰勒展开近似损失函数：

$$\text{Obj} \approx \sum_{i=1}^{N}\left[L(y_i, F_{m-1}(x_i)) + g_i h_m(x_i) + \frac{1}{2} h_i h_m(x_i)^2\right] + \Omega(h_m)$$

其中，g_i 和 h_i 分别是损失函数的一阶和二阶梯度。

3. **最优权重**

对于每个叶子节点 j，其最优权重 w_j^*可以通过最小化目标函数得到：

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda}$$

其中，I_j 是第 j 个叶子节点的样本集合。

5.1.4 XGBoost的特点

- **正则化**：XGBoost在目标函数中加入了正则化项，能够有效防止过拟合。
- **二阶泰勒展开**：通过二阶泰勒展开近似损失函数，加速模型的优化过程。
- **并行计算**：XGBoost支持特征并行和数据并行，能够充分利用多核CPU的计算能力。
- **缺失值处理**：XGBoost能够自动处理缺失值，无需额外预处理。

5.2 优缺点

- **优点:**
  - 在许多任务中表现优异，尤其是在结构化数据上。
  - 支持自定义损失函数，灵活性高。
  - 能够自动处理缺失值，无需额外预处理。
  - 训练速度快，支持并行计算。
- **缺点:**
  - 对参数敏感，需要仔细调参。
  - 训练时间较长，尤其是在数据量较大时。
  - 模型解释性较差，难以理解每棵树的决策过程。

5.3 代码实战

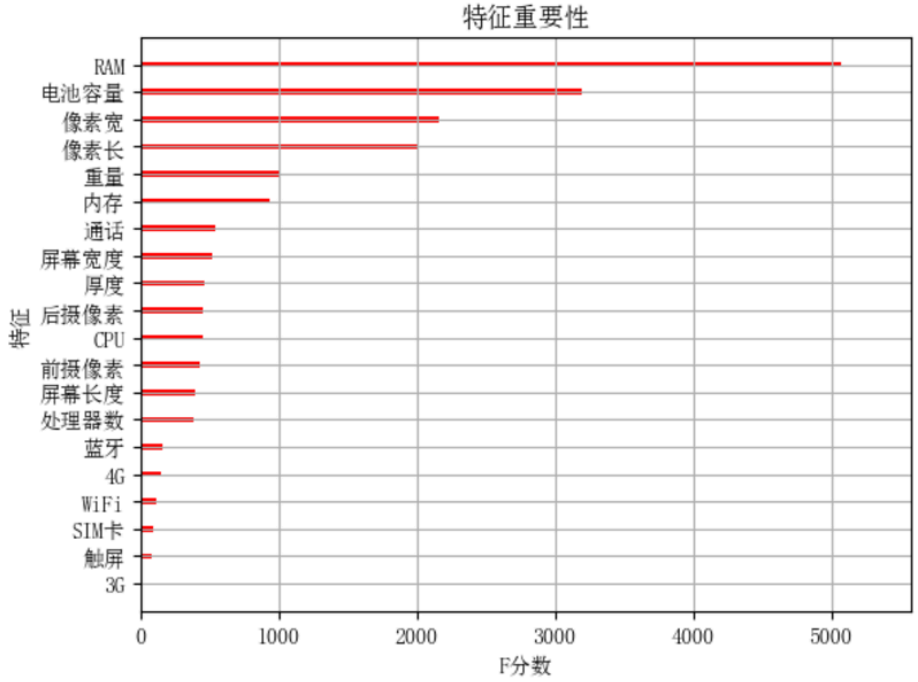
5.3.1 关键参数

- **n_estimators:** XGBoost中树的数量。树的数量越多，模型的性能越好，但计算成本也越高。
- **learning_rate:** 学习率，控制每棵树的贡献程度。较小的学习率需要更多的树来达到相同的性能。
- **max_depth:** 每棵树的最大深度。限制树的深度可以防止过拟合。
- **subsample:** 每轮训练时使用的样本比例。较小的值可以防止过拟合。
- **colsample_bytree:** 每轮训练时使用的特征比例。较小的值可以增加模型的多样性。

5.3.2 基于XGBoost算法的产品定价预测

基于材料中给定的案例，运行XGBoost预测代码，结果如下：

```
***** 决策树 *****
accuracy 0.848
***** 梯度提升树 *****
accuracy 0.906
***** XGBoost *****
accuracy 0.928
```



案例对比了XGBoost和决策树、GBDY，从结果中可以发现，XGBoost算法在三个模型总准确率最好，性能最好。此外，XGBoost算法也可以输出特征重要性，从图中可以看出对结果贡献较高的几个特征，这在一定程度上可以提高XGBoost结果的解释性。

5.4 总结

XGBoost是一种强大的集成学习算法，通过正则化、二阶泰勒展开和并行计算等优化手段，能够在大规模数据集上表现出色。虽然它对参数敏感且训练时间较长，但在许多任务中表现优异，尤其是在结构化数据上。通过合理调参，XGBoost可以在分类、回归和排序任务中取得出色的结果。

6. LightGBM

6.1 原理

LightGBM 是由微软开发的一种基于决策树的梯度提升框架，专为高效处理大规模数据而设计。与 XGBoost 类似，LightGBM 也是一种 Boosting 算法，但它在 XGBoost 的基础上进一步优化了训练速度和内存使用，尤其是在处理高维数据和大规模数据集时表现尤为出色。

6.1.1 LightGBM 的核心思想

LightGBM 的核心思想可以概括为以下几点：

- **基于梯度的单边采样（GOSS）**：通过保留梯度较大的样本，减少计算量。
- **互斥特征捆绑（EFB）**：将互斥的特征捆绑在一起，减少特征数量，从而降低计算复杂度。
- **直方图算法**：将连续特征离散化为直方图，加速特征分裂点的查找。
- **Leaf-wise 生长策略**：与传统的 Level-wise 生长策略不同，LightGBM 采用 Leaf-wise 生长策略，每次选择增益最大的叶子节点进行分裂，从而生成更深的树。

6.1.2 LightGBM的流程

LightGBM 的流程可以总结为以下几个步骤：

1. **初始化模型**：假设初始模型为一个常数，通常为训练集目标值的均值：

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$

其中，$L(y_i, \gamma)$是损失函数，γ是常数。

2. **迭代训练**：对于每一轮$m = 1, 2, \ldots, M$，执行以下步骤：
   - **基于梯度的单边采样（GOSS）**：保留梯度较大的样本，减少计算量。
   - **互斥特征捆绑（EFB）**：将互斥的特征捆绑在一起，减少特征数量。
   - **构建决策树**：使用直方图算法和 Leaf-wise 生长策略构建决策树，选择最优的分裂点，使得目标函数最小化：

$$\text{Gain} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma$$

其中，I_L和I_R分别是分裂后的左子树和右子树的样本集合，λ 和γ是正则化参数。

   - **更新模型**：将新树的结果累加到当前模型中：

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x)$$

其中，ν是学习率，用于控制每棵树的贡献程度。

3. **输出最终模型**：经过M轮训练后，最终的模型为：

$$F(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

6.1.3 算法介绍

1. **基于梯度的单边采样（GOSS）**

LightGBM中的基于梯度的单边采样（Gradient-based One-Side Sampling，GOSS）是一种优化策略，旨在提高梯度提升决策树（GBDT）的训练效率。GOSS的核心思想是在训练每一棵新树时，只使用部分训练数据，而不是全部数据集。具体来说，GOSS会根据样本的梯度（即损失函数关于模型预测的梯度）来选择样本。梯度较大的样本意味着它们对模型的预测误差较大，因此对新树的训练更为重要。GOSS策略会保留一部分梯度较大的样本，并随机选择一部分梯度较小的样本，以减少训练集的大小，从而加快训练速度。

GOSS的过程可以分为两步：

1. 保留一部分具有最大梯度的样本，这些样本对模型训练至关重要，因为它们对损失函数的优化方向有更强的引导作用。
2. 在剩余的梯度较小的样本中，随机选择一部分样本参与训练，并适当放大它们的梯度，以弥补它们在总样本中权重的减少。

通过使用GOSS，LightGBM实际上减少了训练下一个集成树的训练集大小，这将使训练新树的速度更快。同时，通过保留梯度较大的样本并引入少量梯度较小的样本，GOSS能够在不显著损失模型精度的前提下，显著减少参与训练的样本数量，进一步加速模型训练进程。

这种方法可以在不显著损失精度的情况下加快训练速度，特别适用于数据倾斜严重的情况。GOSS通过减少计算量同时保证不会损失太多信息，从而避免了过拟合。在大规模数据集上，GOSS策略能够在不损失过多模型精度的前提下，显著减少参与训练的样本数量，进一步加速模型训练进程。

2. **互斥特征捆绑（EFB）**

互斥特征捆绑（Exclusive Feature Bundling，EFB）是LightGBM中用于处理高维稀疏数据的一种技术。其主要思想是将多个互斥的特征捆绑在一起，形成一个新的特征，从而减少特征的维度，提高训练速度。互斥特征是指在同一行中不会同时出现的特征，例如，在一个分类问题中，每个特征表示一个类别，这些特征在同一行中不会同时为1。

具体来说，EFB的步骤如下：

1. **找到互斥特征**：首先，算法会找到所有的互斥特征。理想情况下，被捆绑的特征都是互斥的，这样特征捆绑后不会丢失信息。
2. **特征捆绑**：然后，算法会将找到的互斥特征捆绑在一起，形成一个新的特征。新的特征的值是原始特征值的和。

这种方法的优点是可以大大减少特征的维度，从而提高训练速度。同时，由于捆绑的是互斥特征，因此不会丢失太多的信息。例如，假设我们有以下的数据集：

特征1	特征2	特征3
1	0	0
0	1	0
0	0	1

在这个数据集中，特征1、特征2和特征3是互斥的，因为它们在同一行中不会同时出现。因此，我们可以将它们捆绑在一起，形成一个新的特征，如下：

新特征
1
1
1

这样，我们就将特征的维度从3降低到了1，大大提高了训练速度。

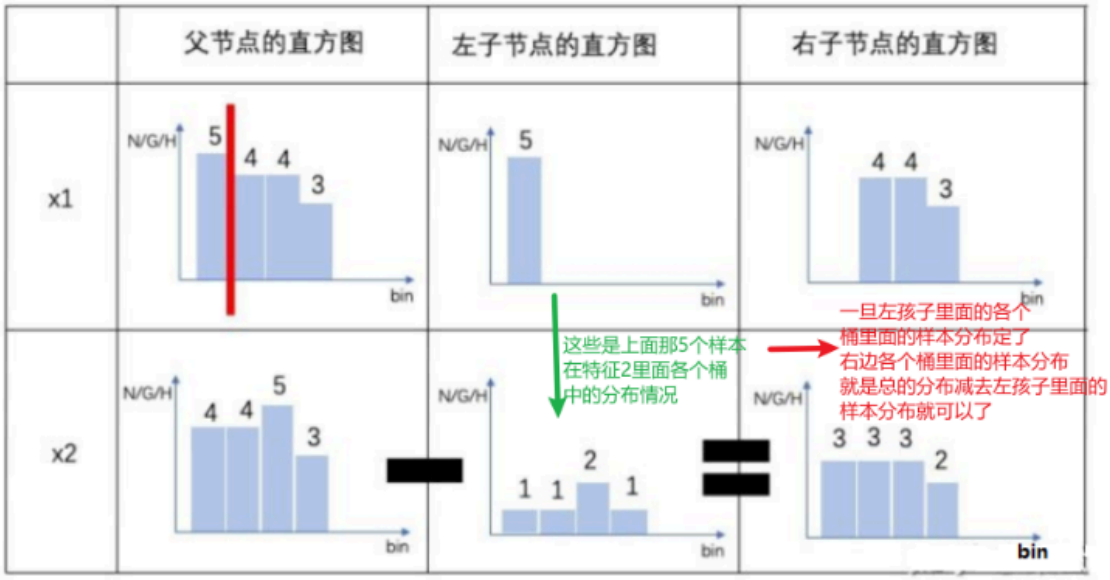
在实际中，不同时非空（简单理解为不同时为1）的特征组合很少，因此LightGBM使用一种基于图着色的近似贪心的特征捆绑策略；图着色问题是一个NP-hard问题，不可能在多项式时间内找到最优解。因此EFB允许特征之间存在少数样本不互斥，这里体现为设定一个最大冲突值K。近似的概念，允许特征簇之间存在一定程度的冲突，进一步的增加可捆绑的特征对 + 降低特征的维度，K的选取也用来平衡准确度和训练效率。

互斥特征捆绑可以通过LightGBM的参数`feature_concurrency`来实现。这个参数的默认值是0，表示不使用特征互斥捆绑。如果将其设置为正整数，则LightGBM将在训练过程中自动检测相关的特征并将它们捆绑在一起。这个参数的值越大，捆绑的特征数量就越多。需要注意的是，特征互斥捆绑有时可能会降低模型的准确性，因此需要谨慎使用。在实际应用中，可以通过交叉验证来确定最佳的`feature_concurrency`参数值。

3. **直方图算法**

LightGBM（Light Gradient Boosting Machine）是一种高效的梯度提升算法，它通过构建多个弱学习器（通常是决策树）来逐步提升模型性能。LightGBM的核心优化之一是直方图算法（Histogram-based Gradient Boosting），该算法通过将连续特征值离散化为有限数量的桶（bins），构建直方图来代替原始数据，从而加快了训练速度并降低了内存占用。

如下图所示，父节点的直方图显示了特征在不同桶（bin）中的样本分布情况，其中每个桶内的样本数量用柱状图表示。左子节点的直方图展示了在特征2上的分布情况，即特征2在不同桶中的样本分布。右子节点的直方图则表示在确定了左子节点的样本分布后，通过从父节点的总分布中减去左子节点的分布，得到右子节点的样本分布。



LightGBM的模型更新公式如下：

$$F(x) = F_0(x) + \nu \sum_{m=1}^{M} h_m(x)$$

其中，$F_0(x)$ 是初始模型，ν 是学习率，$h_m(x)$ 是第 m 棵决策树，M 是树的数量。每棵树的构建过程如下：

1. 构建直方图：将连续特征离散化为直方图。
2. 寻找分裂点：在直方图中寻找最优分裂点，而不需要逐一遍历所有特征的每一个值。
3. 更新模型：将新构建的树添加到模型中，更新模型预测。

通过这些优化，LightGBM在处理大规模数据集和高维特征时表现出色，具有速度快、内存效率高和支持类别特征等特点。

6.1.4 lightGBM的特点

- **高效性**：LightGBM 通过 GOSS 和 EFB 等技术，显著减少了计算量和内存占用。
- **直方图算法**：通过直方图算法加速特征分裂点的查找。
- **Leaf-wise 生长策略**：生成更深的树，提高模型的精度。
- **支持大规模数据**：LightGBM 能够高效处理大规模数据和高维数据。

6.2 优缺点

- **优点:**
  - 训练速度快，内存占用低。
  - 支持大规模数据和高维数据。
  - 在许多任务中表现优异，尤其是在结构化数据上。
- **缺点:**
  - 对参数敏感，需要仔细调参。
  - 模型解释性较差，难以理解每棵树的决策过程。

6.3 代码实战

6.3.1 关键参数

- **n_estimators:** LightGBM 中树的数量。树的数量越多，模型的性能越好，但计算成本也越高。
- **learning_rate:** 学习率，控制每棵树的贡献程度。较小的学习率需要更多的树来达到相同的性能。
- **max_depth:** 每棵树的最大深度。限制树的深度可以防止过拟合。
- **num_leaves:** 每棵树的叶子节点数。较大的值可以提高模型的精度，但也会增加计算量。
- **feature_fraction:** 每轮训练时使用的特征比例。较小的值可以增加模型的多样性。

6.3.2 基于LightGBM算法的中风预测

基于材料中给定的案例，运行LightGBM预测代码，结果如下：

```
LGBM模型准确率：  0.9461839530332681
参数learning_rate的最佳取值:{'learning_rate': 0.01}
LGBM最佳模型得分:0.9522999664413442
[0.95229997 0.95107627 0.9466747 ]
[{'learning_rate': 0.01}, {'learning_rate': 0.05}, {'learning_rate': 0.1}]
参数feature_fraction的最佳取值:{'feature_fraction': 0.8}
LGBM最佳模型得分:0.9488769835562586
[0.9461857  0.94887698 0.9466747 ]
[{'feature_fraction': 0.6}, {'feature_fraction': 0.8}, {'feature_fraction': 1}]
```

初始LightGBM的准确率为0.9462，通过网格搜索和交叉验证，寻找学习率和feature_fraction的最优参数，参数调调优后LightGBM模型的准确率略有升高，为0.9489，说明参数调优过程的有效性。

6.4 总结

LightGBM 是一种强大的集成学习算法，通过 GOSS、EFB 和直方图算法等技术，显著提升了训练速度和内存使用效率。虽然它对参数敏感且模型解释性较差，但在处理大规模数据和高维数据时表现尤为出色。通过合理调参，LightGBM 可以在分类、回归和排序任务中取得出色的结果。

7. 扩展

7.1 各模型对比

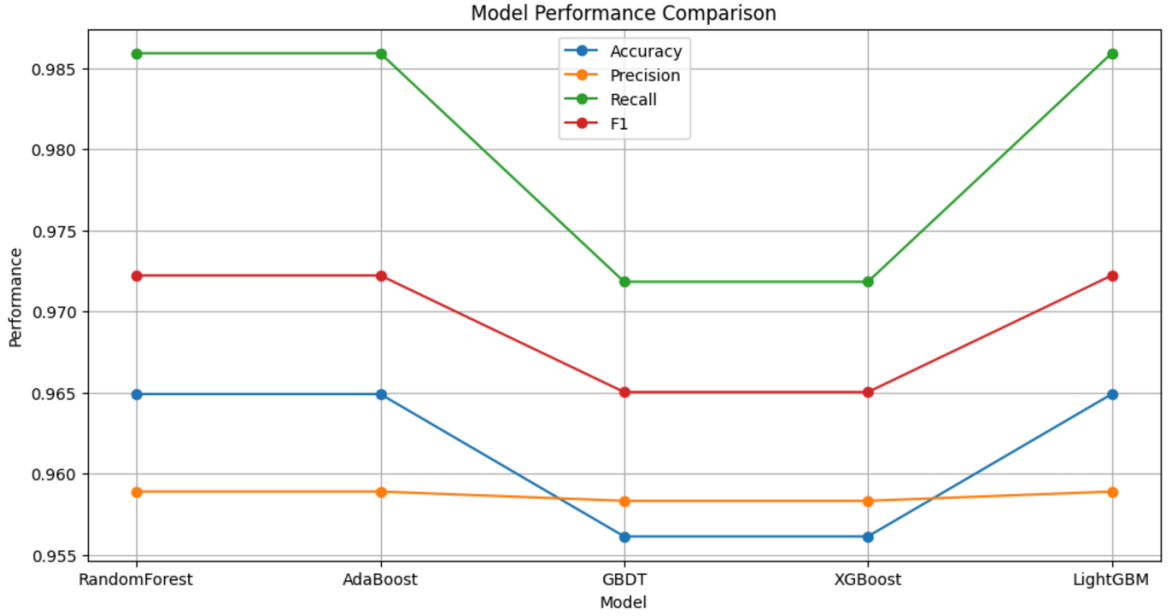
对比随机森林、AdaBoost、GBDT、XGBoost和LightGBM这5个模型的性能，在本次实验中，使用了广泛用于分类任务的 乳腺癌数据集（Breast Cancer Dataset）。该数据集包含了来自临床的乳腺癌患者的多种特征信息，通过这些特征可以预测肿瘤的性质（良性或恶性）。数据集共有 569 条样本和 30 个特征，目标变量为肿瘤的病变情况（良性：0，恶性：1）。

完整代码如下：

```
import warnings
warnings.filterwarnings("ignore")  # 忽略所有警告

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
import xgboost as xgb
import lightgbm as lgb

# 加载数据集
data = load_breast_cancer()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 定义模型
models = {
    "RandomForest": RandomForestClassifier(random_state=42),
    "AdaBoost": AdaBoostClassifier(random_state=42),
    "GBDT": GradientBoostingClassifier(random_state=42),
    "XGBoost": xgb.XGBClassifier(objective="binary:logistic", random_state=42),
    "LightGBM": lgb.LGBMClassifier(random_state=42)
}

# 评估模型
results = {"Model": [], "Accuracy": [], "Precision": [], "Recall": [], "F1": []}
```

```
for name, model in models.items():
    # 训练模型
    model.fit(X_train, y_train)
    # 预测
    y_pred = model.predict(X_test)
    # 评估指标
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    # 记录结果
    results["Model"].append(name)
    results["Accuracy"].append(accuracy)
    results["Precision"].append(precision)
    results["Recall"].append(recall)
    results["F1"].append(f1)

# 将结果转换为DataFrame
results_df = pd.DataFrame(results)
print(results_df)

# 绘制折线图
metrics = ["Accuracy", "Precision", "Recall", "F1"]
fig, ax = plt.subplots(figsize=(12, 6))

for metric in metrics:
    ax.plot(results_df["Model"], results_df[metric], marker='o', label=metric)

ax.set_xlabel("Model")
ax.set_ylabel("Performance")
ax.set_title("Model Performance Comparison")
ax.legend()
plt.grid(True)
plt.show()
```

```
          Model  Accuracy  Precision    Recall        F1
0  RandomForest  0.964912   0.958904  0.985915  0.972222
1      AdaBoost  0.964912   0.958904  0.985915  0.972222
2          GBDT  0.956140   0.958333  0.971831  0.965035
3       XGBoost  0.956140   0.958333  0.971831  0.965035
4      LightGBM  0.964912   0.958904  0.985915  0.972222
```



在乳腺癌数据集上，五个模型的性能对比显示，AdaBoost、LightGBM和RandomForest在准确率、精确率、召回率和F1分数上表现相对一致，均达到了0.964912的准确率和0.958904的精确率，而召回率和F1分数分别为0.985915和0.972222。相比之下，GBDT模型的召回率略高，为0.971831，但准确率和精确率稍低，分别为0.956140和0.958333，F1分数为0.965035。XGBoost的表现相对较弱，准确率、精确率、召回率和F1分数分别为0.956140、0.958333、0.971831和0.965035。总体来看，AdaBoost、LightGBM和RandomForest在各项指标上表现最佳，而XGBoost的表现稍逊一筹。

7.2 集成学习调优方法介绍

7.2.1 交叉验证

K折交叉验证是一种评估模型性能的方法，尤其适用于数据量有限的情况。其步骤如下：

1. **数据分割**：将数据集随机分为K个子集（称为“折”），每个子集大小相近。
2. **模型训练与验证**：
   - 每次使用其中一个子集作为验证集，其余K-1个子集作为训练集。
   - 训练模型并在验证集上评估性能。
3. **重复K次**：每个子集轮流作为验证集，重复K次训练和验证。
4. **性能平均**：将K次评估结果（如准确率、精确率等）取平均，作为模型的最终性能指标。

K折交叉验证的优点是能充分利用数据，减少因数据划分不同导致的性能波动。

使用XGBoost进行K折交叉验证，对比交叉验证前后的模型性能，数据集使用的是sklearn的乳腺癌数据集（二分类任务）。使用5折交叉验证，计算每次验证的性能指标并取平均，完整代码如下：

```
import numpy as np
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import KFold, train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# 加载数据集
```

```
data = load_breast_cancer()
X = data.data
y = data.target

# 划分训练集和测试集（对比交叉验证前后的性能）
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 初始化XGBoost模型
model = xgb.XGBClassifier(objective='binary:logistic', random_state=42)

# 交叉验证前的性能（直接在整个训练集上训练并测试）
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("交叉验证前的性能：")
print("准确率:", accuracy_score(y_test, y_pred))
print("精确率:", precision_score(y_test, y_pred))
print("召回率:", recall_score(y_test, y_pred))
print("F1分数:", f1_score(y_test, y_pred))
print("\n")

# K折交叉验证
k = 5  # 折数
kf = KFold(n_splits=k, shuffle=True, random_state=42)
accuracies, precisions, recalls, f1_scores = [], [], [], []

for train_index, val_index in kf.split(X_train):
    X_train_fold, X_val_fold = X_train[train_index], X_train[val_index]
    y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

    # 训练模型
    model.fit(X_train_fold, y_train_fold)
    y_pred_fold = model.predict(X_val_fold)

    # 记录性能指标
    accuracies.append(accuracy_score(y_val_fold, y_pred_fold))
    precisions.append(precision_score(y_val_fold, y_pred_fold))
    recalls.append(recall_score(y_val_fold, y_pred_fold))
    f1_scores.append(f1_score(y_val_fold, y_pred_fold))

# 输出交叉验证后的平均性能
print("交叉验证后的平均性能：")
print("平均准确率:", np.mean(accuracies))
print("平均精确率:", np.mean(precisions))
print("平均召回率:", np.mean(recalls))
print("平均F1分数:", np.mean(f1_scores))
```

```
交叉验证前的性能：
准确率： 0.956140350877193
精确率： 0.9583333333333334
召回率： 0.971830985915493
F1分数： 0.965034965034965


交叉验证后的平均性能：
平均准确率： 0.9582417582417582
平均精确率： 0.9585977518016883
平均召回率： 0.9749369279147159
平均F1分数： 0.9666205742734195
```

从结果可以发现，交叉验证前，XGBoost模型的准确率、精确率、召回率和F1分数分别为0.9561、0.9583、0.9718、0.9650。5折交叉验证后，XGBoost模型的准确率、精确率、召回率和F1分数分别为0.9582、0.9586、0.9749、0.9666。相较于交叉验证前，交叉验证对于模型性能有一定提升，模型性能更加稳定。

7.2.2 超参数优化

- **网格搜索（Grid Search）**

  网格搜索是一种超参数调优方法，通过遍历所有可能的超参数组合来寻找最优模型配置。其步骤如下：

  1. **定义参数网格**：指定需要调优的超参数及其候选值。
  2. **遍历所有组合**：对每一种超参数组合，训练模型并评估性能。
  3. **选择最优组合**：根据评估结果（如准确率、F1分数等），选择性能最好的超参数组合。

  网格搜索的优点是简单直观，能够找到全局最优解；缺点是计算成本高，尤其是参数空间较大时。

  使用网格搜索调优XGBoost超参数，对比网格搜索前后的模型性能，完整代码如下：

```
import numpy as np
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# 加载数据集
data = load_breast_cancer()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 初始化XGBoost模型
model = xgb.XGBClassifier(objective='binary:logistic', random_state=42)

# 网格搜索前的性能（使用默认参数）
```

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("网格搜索前的性能：")
print("准确率:", accuracy_score(y_test, y_pred))
print("精确率:", precision_score(y_test, y_pred))
print("召回率:", recall_score(y_test, y_pred))
print("F1分数:", f1_score(y_test, y_pred))
print("\n")

# 定义参数网格
param_grid = {
    'max_depth': [3, 5, 7],  # 树的最大深度
    'learning_rate': [0.01, 0.1, 0.2],  # 学习率
    'n_estimators': [100, 200, 300],  # 树的数量
    'subsample': [0.8, 1.0],  # 样本采样比例
    'colsample_bytree': [0.8, 1.0]  # 特征采样比例
}

# 初始化网格搜索
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)

# 执行网格搜索
grid_search.fit(X_train, y_train)

# 输出最优参数
print("最优参数组合：", grid_search.best_params_)

# 使用最优参数训练模型
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)

# 网格搜索后的性能
print("网格搜索后的性能：")
print("准确率:", accuracy_score(y_test, y_pred_best))
print("精确率:", precision_score(y_test, y_pred_best))
print("召回率:", recall_score(y_test, y_pred_best))
print("F1分数:", f1_score(y_test, y_pred_best))
```

```
网格搜索前的性能：
准确率: 0.956140350877193
精确率: 0.9583333333333334
召回率: 0.971830985915493
F1分数: 0.965034965034965


Fitting 5 folds for each of 108 candidates, totalling 540 fits
最优参数组合： {'colsample_bytree': 0.8, 'learning_rate': 0.2, 'max_depth': 3, 'n_estimators': 100, 'subsample': 0.8}
网格搜索后的性能：
准确率: 0.9649122807017544
精确率: 0.958904109589041
召回率: 0.9859154929577465
F1分数: 0.9722222222222222
```

通过结果可以发现，网格搜索能够显著提升模型性能，尤其是在超参数选择对模型影响较大的情况下，通过调优超参数，XGBoost模型的准确率、精确率、召回率和F1分数均有所提高。然而，网格搜索的计算成本较高，适合在小规模参数空间中使用。对于大规模参数空间，可以考虑随机搜索或贝叶斯优化。

- **随机搜索（Random Search）**

  随机搜索是一种超参数调优方法，与网格搜索不同，它通过随机采样超参数组合来寻找最优配置。其步骤如下：

  1. **定义参数空间**：指定需要调优的超参数及其取值范围。
  2. **随机采样**：从参数空间中随机选择一定数量的超参数组合。
  3. **训练与评估**：对每种超参数组合，训练模型并评估性能。
  4. **选择最优组合**：根据评估结果，选择性能最好的超参数组合。

  优点：

  - 计算成本较低，尤其适用于高维参数空间。
  - 能够在较短时间内找到较优的超参数组合。

  缺点：

  - 不一定能找到全局最优解，但通常能找到接近最优的解。

  使用随机搜索调优XGBoost超参数，对比随机搜索前后的模型性能，完整代码如下：

```
import numpy as np
import xgboost as xgb
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# 加载数据集
data = load_breast_cancer()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 初始化XGBoost模型
model = xgb.XGBClassifier(objective='binary:logistic', random_state=42)

# 随机搜索前的性能（使用默认参数）
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("随机搜索前的性能：")
print("准确率:", accuracy_score(y_test, y_pred))
print("精确率:", precision_score(y_test, y_pred))
print("召回率:", recall_score(y_test, y_pred))
print("F1分数:", f1_score(y_test, y_pred))
print("\n")

# 定义参数空间
param_dist = {
    'max_depth': [3, 5, 7, 9],  # 树的最大深度
    'learning_rate': [0.01, 0.05, 0.1, 0.2],  # 学习率
    'n_estimators': [100, 200, 300, 400],  # 树的数量
    'subsample': [0.6, 0.8, 1.0],  # 样本采样比例
```

```
    'colsample_bytree': [0.6, 0.8, 1.0],  # 特征采样比例
    'gamma': [0, 0.1, 0.2],  # 最小损失减少阈值
    'reg_alpha': [0, 0.1, 0.5],  # L1正则化项
    'reg_lambda': [0, 0.1, 0.5]  # L2正则化项
}

# 初始化随机搜索
random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_dist,
                                   n_iter=50, scoring='accuracy', cv=5,
                                   n_jobs=-1, random_state=42, verbose=1)

# 执行随机搜索
random_search.fit(X_train, y_train)

# 输出最优参数
print("最优参数组合：", random_search.best_params_)

# 使用最优参数训练模型
best_model = random_search.best_estimator_
y_pred_best = best_model.predict(X_test)

# 随机搜索后的性能
print("随机搜索后的性能：")
print("准确率:", accuracy_score(y_test, y_pred_best))
print("精确率:", precision_score(y_test, y_pred_best))
print("召回率:", recall_score(y_test, y_pred_best))
print("F1分数:", f1_score(y_test, y_pred_best))
```

```
随机搜索前的性能：
准确率: 0.956140350877193
精确率: 0.9583333333333334
召回率: 0.971830985915493
F1分数: 0.965034965034965


Fitting 5 folds for each of 50 candidates, totalling 250 fits
最优参数组合： {'subsample': 0.6, 'reg_lambda': 0.1, 'reg_alpha': 0.5, 'n_estimators': 300, 'max_depth': 3, 'learning_rate': 0.1, 'gamma': 0.1, 'colsampl
e_bytree': 1.0}
随机搜索后的性能：
准确率: 0.9736842105263158
精确率: 0.9722222222222222
召回率: 0.9859154929577465
F1分数: 0.9790209790209791
```

随机搜索是一种高效的超参数调优方法，尤其适用于高维参数空间。在XGBoost模型中，随机搜索能够显著提升分类性能。对于大规模数据集和复杂模型，随机搜索是比网格搜索更实用的选择。

7.2.3 特征工程

- 特征选择

  特征选择是指从原始数据集中挑选出对模型预测或分类任务最有价值的特征子集，从而去除无关或冗余的特征。其主要目的是提高模型的性能和可解释性，同时减少过拟合的风险。通过统计方法（如卡方检验、互信息法）或基于模型的方法（如基于树模型的特征重要性评分），可以量化每个特征与目标变量之间的相关性或重要性。例如，卡方检验通过衡量特征与目标变量的独立性来筛选特征，而基于树模型的方法则根据特征在树分裂过程中的贡献来评估其重要性。特征选择能够降低模型的复杂度，提高训练效率，并且通过去除噪声特征，使模型更加专注于关键信息，从而在有限的数据上获得更好的泛化能力。

- 特征变换

特征变换是一种通过数学方法对原始特征进行转换或降维的技术，目的是提取更有意义的特征表示，或者降低数据的维度以简化问题。常见的特征变换方法包括主成分分析（PCA）和线性判别分析（LDA）。PCA是一种无监督的降维方法，它通过将原始特征投影到新的正交空间中，使得新的特征（主成分）能够最大程度地保留原始数据的方差信息。LDA则是一种有监督的方法，它不仅考虑数据的分布，还利用类别标签信息，通过寻找能够最大化类间分离度和最小化类内方差的方向进行特征提取。特征变换能够有效减少数据的维度，同时去除冗余信息和噪声，使模型更容易处理高维数据，并且能够提取出更具代表性和区分性的特征，从而提升模型的性能和效率。

除了特征选择和特征变换，特征工程还包括特征构造、特征缩放和特征编码等方法。特征构造是通过对原始数据进行数学运算、组合或利用领域知识生成新的特征，以增强模型对数据的表达能力。特征缩放则通过标准化或归一化处理，将特征值调整到相似的范围，避免数值范围差异对模型训练的影响。特征编码主要用于处理分类特征，例如独热编码（One-Hot Encoding）和标签编码（Label Encoding），将非数值特征转换为数值形式，以便模型能够处理。这些方法与特征选择和特征变换相结合，能够全面提升数据的质量和模型的性能，是机器学习中不可或缺的环节。